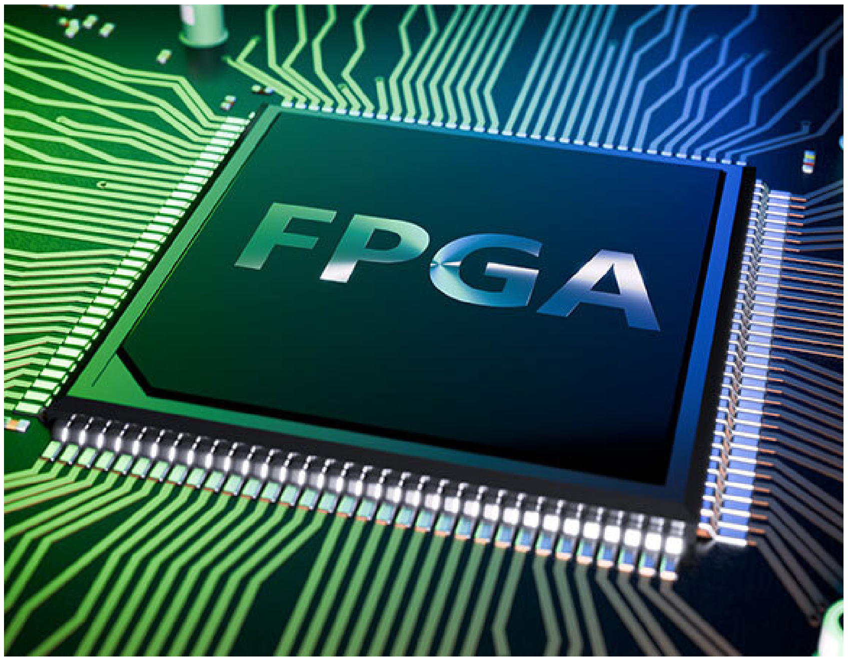
# VHDL User Guide

A reference for VHDL Design



Written by Christian Heimvik

# Contents

**1 Introduction 2**

**2 Constructs 2**
- 2.1 Operators 2
- 2.2 Attributes 3
- 2.3 `map` 4

**3 Fundamental units 4**
- 3.1 `library` 4
- 3.2 `port` 5
- 3.3 `generic` 5
- 3.4 `entity` 6
- 3.5 `component` 7
  - 3.5.1 `configure` 7
- 3.6 `architecture` 8
- 3.7 `package` 9

**4 Object classes 10**
- 4.1 `constant` 10
- 4.2 `variable` 10
- 4.3 `signal` 10
- 4.4 `file` 11

**5 Types 11**
- 5.1 Predefined types 11
  - 5.1.1 `boolean` 11
  - 5.1.2 `bit` 11
  - 5.1.3 `integer` 11
  - 5.1.4 `real` 11
  - 5.1.5 `physical` 12
- 5.2 Custom types 12
  - 5.2.1 `packet` 12
  - 5.2.2 `type` 12
  - 5.2.3 `subtype` 12
- 5.3 Composite types 13
  - 5.3.1 Constrained arrays 13
  - 5.3.2 Unconstrained arrays 13
  - 5.3.3 Multi-dimentional arrays 13
  - 5.3.4 `record` 13

**6 Other things 14**
- 6.1 `assert` 14
- 6.2 Resolution 14

**7 Concurrent code** **15**
7.1 `generate` . . . . . . . . . . 16
7.2 `block` . . . . . . . . . . 16

**8 Sequential code** **17**
8.1 `process` . . . . . . . . . . 18
8.2 `procedure` . . . . . . . . . . 18
8.3 `function` . . . . . . . . . . 19
8.3.1 Functions for overloading . . . . . . . . . . 20

**9 Conventions** **20**
9.1 Flip-flop . . . . . . . . . . 20
9.2 State machines . . . . . . . . . . 21
9.3 Test benches . . . . . . . . . . 22
9.3.1 Clock signal generation . . . . . . . . . . 22

# 1 Introduction

This document provides a comprehensive overview of VHDL and how it can be used. Document is based of the EBNF format of VHDL. In the document, parallel to other programming languages, for the most $\boxed{\|}$ part Python, will be defined in the left margin as shown here. The most obvious parallels (constants, variables, ...) are excluded.

# 2 Constructs

## 2.1 Operators

Operands have to correspond to the operators that are defined for the operand. Various operators inherently in VHDL is given below.

Table 1: VHDL default operators. The difference between a logical and an arithmetic shift is that the new value is 1 in the arithmetic shift, 0 in the logical shift.

| **Operator** | **Description** | **Example (signal assigning)** |
|---|---|---|
| `:=` | Variable assigning | $Z := A + B;$ |
| `<=` | Signal assigning | $Z <= A + B;$ |
| `+` | Addition | $Z <= A + B;$ |
| `-` | Subtraction | $Z <= A - B;$ |
| `*` | Multiplication | $Z <= A \cdot B;$ |
| `/` | Division | $Z <= A/B;$ |
| `mod` | Modulo | $Z <= A$ `modulo` $B;$ |
| `rem` | Remainder | $Z <= A$ `remainder` $B;$ |
| `and` | Bitwise and | $Z <= A$ `and` $B;$ |
| `or` | Bitwise or | $Z <= A$ `or` $B;$ |

| Operator | Description | Example (signal assigning) |
|---|---|---|
| `nand` | Bitwise nand | $Z <= A\ \texttt{nand}\ B;$ |
| `nor` | Bitwise nor | $Z <= A\ \texttt{nor}\ B;$ |
| `xor` | Bitwise xor | $Z <= A\ \texttt{xor}\ B;$ |
| `xnor` | Bitwise xnor | $Z <= A\ \texttt{xnor}\ B;$ |
| `=` | Equality | $Z <= A = B;$ |
| `/=` | Inequality | $Z <= A \neq B;$ |
| `<` | Less than | $Z <= A < B;$ |
| `<=` | Less than or equal | $Z <= A <= B;$ |
| `>` | Greater than | $Z <= A > B;$ |
| `>=` | Greater than or equal | $Z <= A \geq B;$ |
| `not` | Logical not | $Z <= \neg A;$ |
| `shift left` | Shift left | $Z <= A \ll B;$ |
| `shift right` | Shift right | $Z <= A \gg B;$ |
| `rol` | Rotate left | $Z <= \texttt{rol}(A, B);$ |
| `ror` | Rotate right | $Z <= \texttt{ror}(A, B);$ |
| `sll` | Shift left logical | $Z <= \texttt{sll}(A);$ |
| `srl` | Shift right logical | $Z <= A\ \texttt{srl}\ B;$ |
| `sla` | Shift left arithmetic | $Z <= A\ \texttt{sla}\ B;$ |
| `sra` | Shift right arithmetic | $Z <= A\ \texttt{sra}\ B;$ |
| `abs` | Absolute value | $Z <= \texttt{abs}\ A;$ |

Note that the operator has to be defined for its operands.

## 2.2 Attributes

An attribute returns a value containing information about a data vector. In python this would be functions called on lists to give information about the list' properties.

Table 2: Different attributes to an array `x` and what it returns. Note that some of these, especially the search variants are not so easily synthesizable.

| Attribute | Description | Example |
|---|---|---|
| `x'left` | Returns the leftmost index of the range | `x'left = 2` |
| `x'right` | Returns the rightmost index of the range | `x'right = 0` |
| `x'low` | Returns the lowest array index | `x'low = 0` |
| `x'high` | Returns the highest array index | `x'high = 2` |
| `x'range` | Returns the range of the array or vector | `x'range = 2 downto 0` |
| `x'length` | Returns the number of elements in the range | `x'length = 3` |
| `x'ascending` | Returns true if the range is ascending | `x'ascending = false` |
| `x'delayed(T)` | Returns a signal delayed by time `T` | `y <= x'delayed(10 ns);` |
| `x'stable(T)` | True if signal `x` has not changed in time `T` | `if (x'stable(5 ns)) then` |
| `x'quiet(T)` | True if no event has occurred on `x` in time `T` | `if (x'quiet(5 ns)) then` |
| `x'last_event` | Time since last event on signal `x` | `if (x'last_event > 10 ns) then` |

| Attribute | Description | Example |
|---|---|---|
| `x'last_value` | Last value of signal `x` before the current event | `y <= x'last_value;` |
| `x'active` | True if `x` is active (usually '1') | `if (x'active) then` |
| `x'event` | True when an event has occurred on signal `x` | `if (x'event) then` |
| `x'pos(V)` | Returns the position number of value `V` | `x'pos(1) = 2` |
| `x'val(P)` | Returns the value at position `P` | `x'val(2) = '1'` |
| `x'succ(V)` | Returns the successor of value `V` | `x'succ('0') = '1'` |
| `x'pred(V)` | Returns the predecessor of value `V` | `x'pred('1') = '0'` |
| `x'leftof(V)` | Returns the value to the left of `V` | `x'leftof('1') = '0'` |
| `x'rightof(V)` | Returns the value to the right of `V` | `x'rightof('0') = '1'` |
| `A'left(x)` | Returns the value at the left border in dimension `x` | `A'left(1) = 1` |
| `A'right(x)` | Returns the value at the right border in dimension `x` | `A'right(1) = 4` |
| `A'range(x)` | Returns the range in dimension `x` | `A'range(1) = 1 to 4` |
| `A'length(x)` | Returns the length in dimension `x` | `A'length(1) = 4` |

## 2.3 `map`

The map after a generic or port statement connects specific values or signals to the placeholders defined in an entity or component. This ensures that the design's inputs and outputs are correctly linked to their intended connections.

- **Syntax:**

```
... map (*from signal/variable* => *to signal/variable*)
```

- **Example:**

```
generic map (width => 32)
port map (in1_uut => out1_tb)
```

# 3 Fundamental units

## 3.1 `library`

The `library` statement defines a library to access VHDL packages and functions. However, objects from libraries are not automatically visible; explicit inclusion with the `use` clause is required to access specific items. Without the `use` clause, even if the library is defined, its contents are not visible.

- **Syntax:**

```
library *library name*\{...\}
use *library name*.(identifier | all) ;
```

- **Example:**

```
library ieee;
use ieee.std_logic_1164.all;
```

## 3.2 port

Ports define how the entity communicates with other components or systems, specifying the types of signals it can accept or produce. This is analogous to specifying of which types each of the parameters has in C or C++.

- **Syntax**:

```
port ( port_name : mode data_type );
```

  where the different modes are: `in`, `out`, `inout`, `buffer`. **NB: `inout` requires resolution.**[1]

- **Example**:

```
port(in1, in2 : in std_ulogic := '1';
     twoway : inout std_logic := 'Z';
     buffout : buffer std_ulogic;
     out1 : out std_ulogic);
```

## 3.3 generic

Enables parameter-passing in entities. This makes the entity (the interface of our circuit to the externals) flexible and reusable. Think of the entity as the function itself, and the parameter list in Python is the generics in VHDL.

- **Syntax:**

```
generic(*name of signal/variable* : *type*)
```

- **Example:**

```
--Entity definition:
entity reg is
    generic(width:positive);
    port(d : in std_logic_vector(0 to width-1);
        q : out std logic_vector(0 to width-1);
        clk,rst : in std_logic);
end entity reg;

--*Architechture definition*

--Use in another architecture:
WIDE_REG: entity work.reg(behaviour)
    generic map (width <= 32);
    port map ( q => wide_out, d=> wide_in, rst => reset, clk => clock);
```

[1]In VHDL, **resolution** refers to the process of determining the final value of a signal when multiple sources drive it simultaneously, typically using a predefined or user-defined resolution function to manage conflicts, especially for signals with the `inout` mode. Further discussed in section 6.2.

## 3.4 `entity`

Defines the external interface of a digital component or module. It specifies the inputs, outputs, and any other ports of the module but does not include the internal implementation details. Think of this as a parameter list to our function in python, specifying which way each of the parameters take.

- **Syntax:**

```
-- Definition:
entity *entity_name* is
    generic (*parameter_list*);
    port (*port_list*);
end entity *entity_name*;

-- Architecture:
architecture *architecture_name* of *entity_name* is
begin
    -- Behavioral or structural description of the entity
    -- Concurrent statements
end architecture *architecture_name*;

-- Instantiation:
*label*: entity work.*entity_name*(*architecture_name*)
    generic map (*parameter_mapping*)
    port map (*port_mapping*);
```

- **Example:**

```
-- Definition:
entity xor_gate is
    port (
        in1, in2 : in std_logic;
        out1 : out std_logic
    );
end entity xor_gate;

-- Architecture:
architecture behavior of xor_gate is
begin
    out1 <= in1 xor in2;  -- Simple XOR gate functionality
end architecture behavior;

-- Instantiation:
architecture behavior_tb of xor_tb is
    signal signal1, signal2 : std_logic;
    signal result : std_logic;
begin
    DUT: entity work.xor_gate(behavior)
```

```
        port map (
            in1 => signal1,
            in2 => signal2,
            out1 => result
        );
end architecture behavior_tb;
```

### 3.5 `component`

The `component` statement is something that we can use to modularize only the **instantiation** of an entity. The `component` statement does not affect the entity definition, nor the architecture defining the functionality of the entity.

- **Syntax:**

```
--Definition:
component *component name* is
    generic (*parameter list*);
    port (*port list*);
end component *component name*

--Instantiation:
*label*: *component name*
    generic map (*parameter mapping*)
    port map (*port mapping*);
```

- **Example:**

```
architecture test_xor of test_bench is
    signal In1, In2, Out1, Out2 : bit;
    component xor_comp is
        port(In1, In2 : in bit; Out1 : out bit);
    end component xor_comp;
begin
DUT : component xor_comp
    port map( In1, In2, Out1 );
end architecture test_xor;
```

#### 3.5.1 `configure`

However, to tie the component to the already defined and functional defined architecture of the entity, we must configure the component we the entity to the component using the `configuration` statement.

- **Syntax:**

```
configuration *configuration name* of *calling entity name* is
    for *calling architecture name*
```

```
        for *label* : *callee component name*
            use entity work.*callee entity name*;
        end for;
    end for;
end configuration *configuration name*;
```

- **Example:**

```
configuration test_config of test_bench is
    for test_xor
        for DUT : xor_comp
            use entity work.xor_gate(structure);
        end for;
        for others : xor_comp
            use entity work.xor_gate(behavior);
        end for;
    end for;
end configuration test_config;
```

The configuration may be done `outside` the upper level architecture.

### 3.6 `architecture`

Architecture describes the implementation of the entity (or the interface). It contains the actual logic and behavior of the module and is analogous to the actual content of the function in python.

- **Syntax:**

```
architecture *architecture name* of *entity name* is
    --Declarations
begin
    --Concurrent statements
end architecture_name;
```

  Declarations can be `signals`, `variables`, `constants`, `types`, `subtypes`, `components` or `files`. Concurrent statements can be whatever happens concurrently, as long as it defines the behaviour of the entity (which is the purpose of the architecture).

- **Example:**

```
architecture structure of xor_gate is
    signal internal1, internal2: std_ulogic;
begin
    U0: entity work.and_gate(behavior)
        port map (in1, in2, internal1);
    U1: entity work.nor_gate(behavior)
        port map (in1, in2, internal2);
    U2: entity work.nor_gate(behavior)
```

```
        port map (internal1, internal2, out1);
end architecture structure ;
```

In the example, 3 components U0, U1, and U2 gets instantiated concurrently to implement the different entities.

## 3.7 package

A `package` is used to group related declarations, such as types, constants, signals, components, functions, and procedures, into a single, reusable unit. This modular approach enhances code organization, reusability, and maintainability. Packages can be compiled separately and then used in different parts of a VHDL design by referencing them with the `use` statement.

- **Syntax:**

```
--Package definitions
package *package name* is
    --Declarations
end package *package name*;

--Package implementation
package body *package_name* is
    --Implementations of functions and procedures
end package body *package_name*;
```

- **Example:**

```
-- Package Header
package My_Package is
    constant WIDTH : integer := 8;
    type My_Array is array (0 to WIDTH-1) of std_logic;
    function Add_One (A: integer) return integer;
end package My_Package;

-- Package Body
package body My_Package is
    function Add_One (A: integer) return integer is
    begin
        return A + 1;
    end function Add_One;
end package body My_Package;

-- Using the package in an architecture
library IEEE;
use IEEE.std_logic_1164.all;
use work.My_Package.all;

```

```
architecture Behavioral of MyModule is
    signal Result : integer;
begin
    Result <= Add_One(3); -- Calls the function from the package
end Behavioral;
```

## 4 Object classes

### 4.1 constant

Permanently defined values that remain unchanged throughout the simulation.

- **Syntax:**

```
constant *name* : *data_type* := *value*;
```

- **Example:**

```
constant WIDTH : integer := 8;
```

### 4.2 variable

Values that can change immediately within a process. Try to make the intermediates as much as you can, as they make the simulation faster, as no $\Delta$-delay is present. Are only available **inside processes.** When assigned, the variable changes **immediately**.

- **Syntax:**

```
variable *name* : *data_type* := *initial_value*;
```

- **Example:**

```
variable count : integer := 0;
```

### 4.3 signal

Values that change after a delay or explicitly specified timing. When assigned, the assignment will be added to the event queue, where the variable will be assigned to the value the assigner has **now**, but will be assigned **after** the process is suspended.

- **Syntax:**

```
signal *name* : *data_type* := *initial_value*;
```

- **Example:**

```
signal clk : std_logic := '0';
```

### 4.4 `file`

Used for long-term storage and retrieval of data, loaded at runtime and written to during simulation.

- `Syntax:`

```
file *file_name* : text open *mode* is "*file_path*";
```

- **Example:**

```
file my_file : text open write_mode is "output.txt";
```

# 5 Types

## 5.1 Predefined types

#### 5.1.1 `boolean`

- **Description:** A result of an evaluation of `=`, `/=`, `<`, `<=`, `>` or `>=`.
- **Example:**

```
chip_en := ( address_bus = chip_id );
```

#### 5.1.2 `bit`

- **Description:** A result of an evaluation of `and`, `or`, `nand`, `nor`, `xor`, `xnor` or `not`.
- **Example:**

```
'0' and '1' = '0'
```

#### 5.1.3 `integer`

An integer.

- **Example:**

```
signal *signal_name* : integer;
variable *variable_name* : integer;
```

#### 5.1.4 `real`

A 32-bit floating point with mantissa and exponent. **NB: Not easily synthesizable!**

- **Example:**

```
signal *signal_name* : real;
variable *variable_name* : real;
```

#### 5.1.5 `physical`

Value with denomination.

- **Example:**

```
physical *type_name* is *unit*;
```

## 5.2 Custom types

### 5.2.1 `packet`

Different types bundeled tougether into one. Analogous to `structs` in C and C++.

- **Example:**

```
package Numbers is
    type small_number_A is range 0 to 255;
    type small_number_B is range 0 to 255;
    subtype byte_1 is small_number_A range 0 to 7;
end package;

use work.Numbers.all;
```

### 5.2.2 `type`

A custom type definition.

- **Example:**

```
type alu_func is (add, subtract, mult);
type octal_digit is (’0’,’1’, ’2’, ’3’, ’4’, ’5’, ’6’, ’7’);

variable alu_op : alu_func;
variable last_digit : octal_digit := ’0’;
```

For this to work with operators, operators for this newly given type will have to be defined.

### 5.2.3 `subtype`

Based on restricted subset of the values of an already defined type.

- **Example:**

```
subtype ibyte is integer range 0 to 255;
subtype probability is real range 0.0 to 1.0;
```

The subtype inherits the operators off the type it is based off. Similar to the `typedef` declaration in C.

## 5.3 Composite types

### 5.3.1 Constrained arrays

Indexed collection of data of the same type. One or more dimentions avalibale.

- **Example:**

```
type be_word is array (0 to 15) of bit; --big endian
type le_word is array (15 downto 0) of bit; --little endian

signal buffer_register : be_word := (0 =>'1',1 =>'0,2 to 15 => '0');
signal buffer_register : be_word := (5, 20, 0, 6, 23);
--may also be variable or constant
```

### 5.3.2 Unconstrained arrays

Only declare the type of the index-value of an array, but not the range. Upon initializing the instance of the array we give it a length. It is only the **type** of array that is unconstrained, the different implementations are constrined.

- **Example:**

```
type sample s array (natuaral range <>) of integer --unconstained
-- may also be std_logic_vector, signed and unsigned

subtype long_sample is sample(0 to 63); --init to constrained
```

### 5.3.3 Multi-dimentional arrays

An array that has several dimentions. Indexing is **row-major**.

- **Example:**

```
type matrix is array (0 to 3, 0 to 3) of real;
signal picture : matrix :=
(0 => (0 => 0.0, others => 4.0),
1 => (1 => 1.0, others => 5.0),
2 => (2 => 2.0, others => 6.0),
3 => (3 => 3.0, others => 7.0));
```

### 5.3.4 `record`

Combination of elements that may have different types. **NB: Not synthesizable!** But still may be applicable in testbenches.

- **Example:**

```
type time_stamp is record
min : integer range 0 to 59;
hour : integer range 0 to 23;
end record time_stamp;
```

## 6 Other things

### 6.1 `assert`

The `assert` statement is used to check a boolean expression and can optionally report an expression and specify a severity level if the condition is not met. Assertions help detect errors and provide meaningful messages during **simulation**. Synthesis tools often ignore assertions, but they can be used for simulation verification.

- **Syntax:**

```
assert boolean_expression
--report expression
--severity expression;

--severity levels are: note, warning, error, failure
```

- **Example Usage:**

```
-- Simple assertion without report and severity
assert current_value <= max_value;

-- Assertion with a report message
assert current_value <= max_value
report "current_value too large";

-- Assertion with a report message and severity level
assert current_value <= max_value
report "current_value too large"
severity warning;
```

### 6.2 Resolution

The resolution mechanism allows multiple drivers to control the same signal. The designer must specify how to resolve the value of the signal when driven by multiple sources.
A resolution function is used to specify how to resolve the signal value when multiple drivers are present. It takes an array of values and returns a single resolved value according to predefined rules.

- **Example:**

```vhdl
type simple_logic is ('0', '1');
type simple_logic_array is array (integer range <>) of simple_logic;
subtype std_simple_logic is resolve_simple_logic simple_logic;

function resolve_simple_logic (values : in simple_logic_array) return simple_logic
begin
    -- Assume default value is '0'
    return '0';
end function resolve_simple_logic;

--In signal declaration:
signal s1 : resolve_simple_logic simple_logic;
signal s2 : std_simple_logic;
--Both are resolved
```

## 7 Concurrent code

All code lines in an `architecture` is executed concurrently (each time a signal on the right side changes value). If a signal changes value, the value will be transferred to the other side after Δ-delay. All constructions in section 7 are constructions that are **only** valid in a concurrent environment. VHDL has 3 different types of concurrent assignments.

- **Regular**

  Regular assignment with either `:=` or `=>`.

  - **Example:**

  ```vhdl
  cs <= a xor b;
  cv := a xor b;
  ```

- **Conditional**

  Assignment based on some conditions. Parallel to assignment by `if`-statements (but concurrently) in python.

  - **Example:**

  ```vhdl
  cout <= '1' when a = '1' and b = '1' and cin = '0' else '0';
  ```

- **Selected**

  One of may alternatives is selected. Parallel to assignment by `select-case` in python.

  - **Example:**

```
eval <= a & b & cin;
with (eval) select cout <=
'1' when '110',
'1' when '101',
'1' when '111',
'0' when others;
```

### 7.1 `generate`

Generate is used to create multiple instances of a block of code, typically to instantiate components, processes, or concurrent statements repeatedly. Can only be used in concurrent regions of the code. We have two types of generate-statements:

- `for ... generate`

  The `for-generate` statement allows you to repeat a block of code a specified number of times. Each iteration of the code is evaluated concurrently.

  - **Example:**

```
    gen: for i in 0 to 3 generate
    C(i) <= A(i) and B(i);
    end generate;
```

- `if ... generate`

  The `if-generate` statement allows for the conditional generation of code blocks.

  - **Example:**

```
    gen: if USE_REG generate
        Q <= D when rising_edge(clk);
    end generate;
```

### 7.2 `block`

The block statement in VHDL groups concurrent statements and provides a local scope for signals, variables, and constants. It is used for modular design and hierarchical organization.

- **Example:**

```
architecture Behavioral of SimpleBlock is
    signal A, B, C: std_logic;
begin
    my_block: block
    begin
        C <= A and B;
    end block my_block;
end Behavioral;
```

## 8 Sequential code

Code that runs sequentially needs to be placed into a process. The following constructions are **only** valid in a sequential environment.

- `if`

```
if a = '1' then
    b <= '1';
else
    b <= '0';
end if;
```

- `case`

```
case i is
    when 0 => c <= '0';
    when 1 => c <= '1';
    when others => c <= 'Z';
end case;
```

  Make sure to include `when others` to ensure we are covering all cases.

- `while`

```
while i < 5 loop
    i := i + 1;
end loop;
```

- `for`

```
for j in 0 to 3 loop
    a <= not a;
end loop;
```

- `wait`

```
wait on A;
wait until rising_edge(A);
wait for 10 ns;
```

  On a `wait` statement, the process gets suspended, and only woken up again if triggered. We can only have one wait per process.

The expression `next` jumps to the next iteration of the loop. The expression `exit` exits the loop.

## 8.1 process

A process itself may be defined in a concurrent environment, **but within the process, the code is executed sequentially**. Processes can be either implicit or explicit.

- **Implicit process**

  When no sensitivity is explicitly defined.

  - **Example:**

```
internal1 <= in1 and in2;
```

- **Explicit process**

  When we declare a process, and thus its sensitivity, explicitly. Here we must define its sensitivity explicitly to enable the triggering. Everything inside an explicit process is executed sequentially, but concurrently with other concurrent statements outside the process.

  - **Syntax:**

```
*label*: process *list of signals to sense* is
-- Declarations
begin
-- Sequential statements
end process *label*;
```

  - **Example:**

```
exProcess: process (C,D) is
signal A,B : std_logic;
begin
    A <= C;
    B <= D;
    X <= A xor B
    --Both executed sequentially
end process exProcess;
```

  One could also do `wait on C, D, A, B;` to restart the process to make sure something happens on a change of any of these.

## 8.2 procedure

A procedure encapsulates a number of sequential statements, without returning anything. It is like a function in python without any return value.

- **Syntax:**

```
procedure *procedure name* (*parameter list*) is
    --Declarations
begin
    --Sequential statements
end procedure *procedure name*;
```

- **Example:**

```
architecture behavior of entity_test is
    signal A : integer := 1;signal B:integer:= 2; signal C:integer := 4;
    signal D,E : integer;
    procedure parameter_test(signal s_in:in integer;signal s_out: out integer) is
    begin
        ut <= A + inn;
    end procedure parameter_test;
begin
    parameter_test( B, D );
    parameter_test( C, E );
end architecture behavior;
```

Note that without a resolution function, two or more procedures cannot drive the same signal. Also note that a procedure is executed sequentially, and a call to a procedure in a concurrent environment such as this

```
    parameter_test( B, D );
```

, is interpreted as this

```
call_proc : process is
begin
    parameter_test( B, D );
    wait on B;
end process call_proc;
```

to ensure true sequential execution.

## 8.3 function

A function also encapsulates several sequential statements but returns a result opposing the procedure. It therefore must contain a `return` statement, and should be pure.

- **Syntax:**

```
function *function name* (*parameter list*) return *returntype* is
    --Declarations
begin
    --Sequential statements
end function *function name*;
```

- **Example:**

```vhdl
function and_gate (a:std_logic; b : std_logic) return std_logic is
begin
    return a and b;
end function and_gate;

--Usage:
architecture Behavioral of MyModule is
begin
    C <= and_gate(A, B);
end Behavioral;
```

### 8.3.1 Functions for overloading

Function overloading in VHDL allows you to define multiple functions with the same name but different parameter types. This feature enables more intuitive and flexible use of functions by adapting their behavior based on the types of the input arguments.

- **Example:**

```vhdl
function "+"(A: integer; B: bit_vector) return signed is
    -- Function implementation here
end function "+";

architecture Behavioral of Entity_Test is
    signal Int1, Int2, Int3 : integer;
    signal Bv : bit_vector;
    signal Signe : signed;
begin
    Int3 <= Int1 + Int2; -- Uses the built-in "+" for integers
    Signe <= Int1 + Bv;   -- Uses the overloaded "+" function
end architecture Behavioral;
```

# 9 Conventions

## 9.1 Flip-flop

The convention of code for a (synchronous) flip-flop is as follows.

```vhdl
sync: process (clk) is
begin
    if (clk'event and clk='1') then
        if reset = '1' then
            q <= '0';
        else
```

```
            q <= d;
        end if;
    end if;
end process;
```

## 9.2 State machines

We have the two types of state machines:

- **Moore Machine**: Outputs depend only on the current state. Outputs are synchronous.
- **Mealy Machine**: Outputs depend on the current state and the current input. Outputs are asynchronous.

When designing state machines one should

- Split the state machine into two processes: combinatorial and sequential.
- Use enumerated state vector for the state (possibly with defined Grey-decoding).
- Define a default state.
- Use reset of flip-flops to a known state.

Use the following convention.

```
entity fsm is
    port (
        i_sig : in std_logic;
        rst : in std_logic;
        clk : in std_logic;
        o_sig : out std_logic);
    end entity fsm;

architecture rtl of fsm is
    type state is (IDLE, ONE, TWO, THREE);
    signal curr_state, next_state : state;
begin
    CombProc : process (i_sig, curr_state)
        begin
            case (curr_state) is
                when IDLE =>
                    --Setting NEXT STATE and OUTPUTS in this state
                when others =>
                    --Setting NEXT STATE and OUTPUTS other states
                    --MUST BE INCLUDED!
            end case;
        end process CombProc;

    SynchProc : process (rst, clk)
        begin
            --Setting CURRENT_STATE at edges or asynchronously
        end if;
    end process SynchProc;

end architecture rtl;
```

Storing data can be done in a state machine by adding a shift register and storing the desired data by control signals to the shift registers in each state.

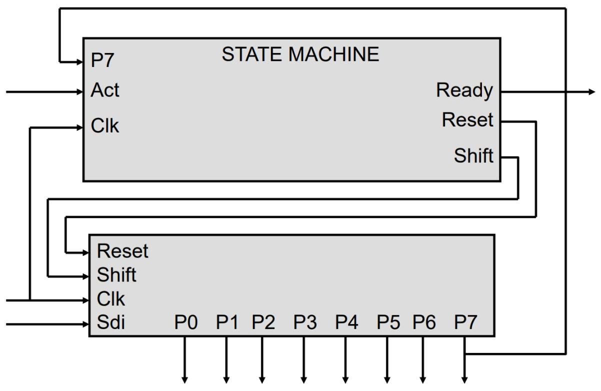


Figure 1: The figure shows how one can control an data storage unit with the control signals of an FSM, rather than storing the data in the state bits themselves.

## 9.3 Test benches

Each test bench should instantiate the circuit it should test. Can apply stimuli from following sources.

- Generated stimuli directly in the test banch
- Read vectors from array
- Read vectors from file

And compare with the expected results. Possible, and desired to use two different versions of the architecture under test, and test both with outputs compared in comparator.

### 9.3.1 Clock signal generation

Can be done by either doing

```
ClockFast <= not ClockFast after FastClockPeriod/2;
```

or

```
process begin
wait for (SlowClockPeriod * 0.6);
ClockSlow <=    1   ;
wait for (SlowClockPeriod * 0.4);
ClockSlow <=    0   ;
end process;
```